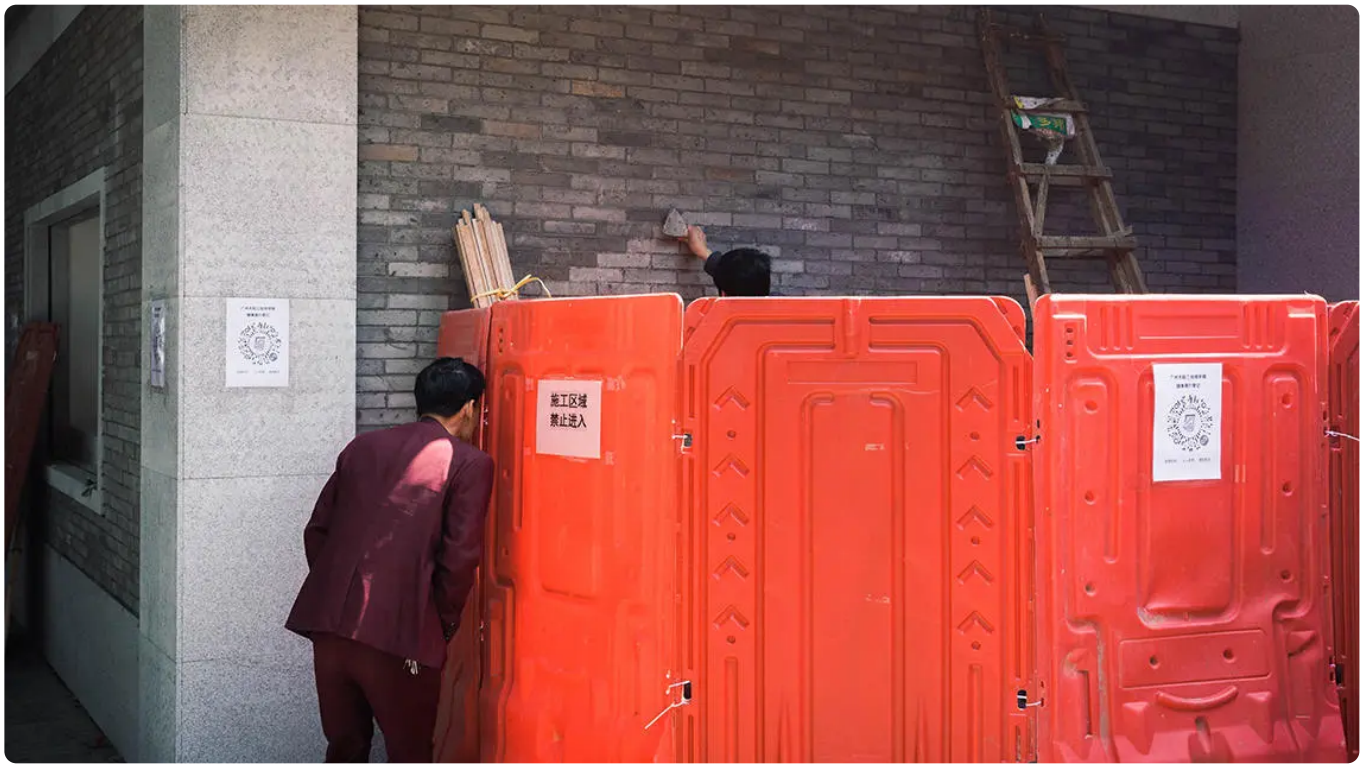
下载APP

33｜限流算法：如何防止系统过载？

2022-03-10 黄清昊

《算法实战高手课》 课程介绍 >



讲述：黄清昊

时长 16:10 大小 14.81M

你好，我是微扰君。

上一讲我们学习了业务场景中频繁会使用到的延时队列，能帮助处理很多业务上的定时任务问题，因为这个组件的功能和具体业务往往没有关系，我们通常会利用各种中间件来实现延时队列的能力。

领资料

今天我们来探讨另外一个算法的原理和实现，它也和业务本身没有强关联，但是在各个业务场景下都非常常见，那就是限流算法。

限流算法，也被我们常称为流控算法，顾名思义就是对流量的控制。日常生活中有很多例子，比如地铁站在早高峰的时候，会利用围栏让乘客们有序排队，限制队伍行进的速度，

避免大家一拥而上，这就是一种常见的限流思路；再比如在疫情期间，很多景点会按时段限制售卖的门票数量，避免同一时间在景区的游客太多等等。

这些真实生活中的方案本质都是因为在某段时间里资源有限，我们需要对流量施以控制。其实同样的，**在互联网的世界里，很多服务，单位时间内能承载的请求也是存在容量上限的，我们也需要通过一些策略，控制请求数量多少，实现对流量的控制**。

当然在工程中，“流量”的定义也是不同的，可以是每秒请求的数量、网络数据传输的流量等等，所以在不同的场景下，我们也需要用不同的方式限制流量，以保证系统不至于被过多的流量压垮。虽然，限流一定会导致部分请求响应速度下降或者直接被拒绝，但是相比于系统直接崩溃的情况，限流还是要好得多。

业务中的限流场景

好，现在相信你明白限流非常重要了，那在我们的开发中有哪些场景需要考虑限流呢？

一个非常典型的例子就是之前在 TCP 中提到的拥塞控制算法，这可以被认为是一种经典的限流算法，其中借助窗口控制流量的思路也是限流算法中非常常用的。

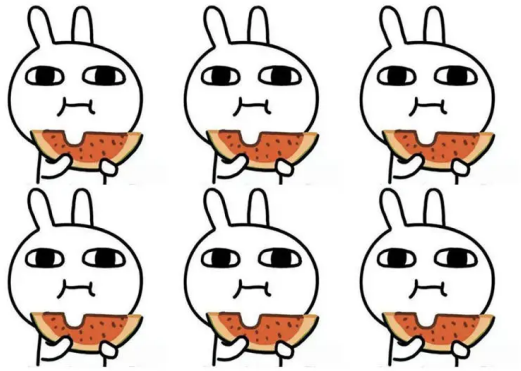
但是对于我们大部分业务开发工程师来说，肯定不会去修改 TCP 协议了，还有哪些典型的业务中的限流场景呢？

最常见的就是三种情况：

- 突发流量
- 恶意流量
- 业务本身需要

突发流量

先放两张图，相信你看一眼就知道突发流量要说什么了。



不明真相的吃瓜群众

双十一快到了

没错，我们最经常听到的服务器大规模崩溃的场景就是两个：“某个突发新闻出现，微博平台又崩了”、“双十一来了，买买买，诶，某电商平台出现故障”。

这正是我们需要限流的主要场景之一。面对业务流量突然激增，因为后端服务处理能力有限，遇到突发流量时，很容易出现服务器被打垮的情况。

在这个情况下，除了提供更好的弹性伸缩的能力，以及在已经能预测的前提下提前准备更多的资源，我们还能做的一件事就是利用限流来保护服务，即使拒绝了一部分请求，至少也让剩下的请求可以正常被响应。

恶意流量

除了突发流量，限流有的时候也是出于安全性的考虑。网络世界有其凶险的地方，所有暴露出去的 API 都有可能面对非正常业务的请求。

比如各种各样的爬虫，或者更直接的恶意攻击，都可能会在很短的时间里，大规模的疯狂调用我们服务对外暴露的接口，这同样可能导致服务崩溃，在很多时候也会导致我们需要的计算成本飙升，比如云计算的场景下。

业务本身需要

第三种情况也很常见，就是业务本身的需要。

最典型的，现在很多云产品都会推出不同等级的服务：

云产品业务，本身就会对每个客户使用产品流量的情况进行限制，客户采购等级越低的服务，支持的 TPS 数量就会相对来说低一些。这不只是出于成本的考虑，也出于商业利益的考虑。

总的来说，这三种情况，都需要有一个有效的流量控制算法，来支持我们的需求。那如何才能准确的限制流量呢？这就是我们的限流算法要考虑的问题了。

如何准确限流

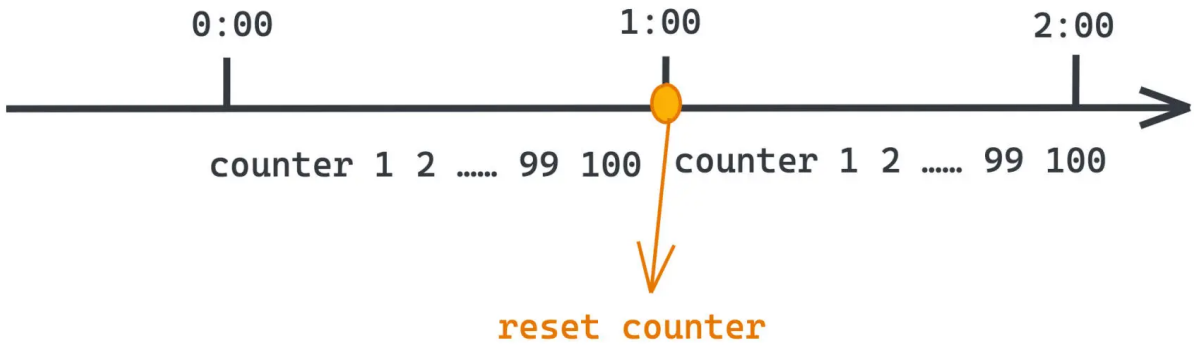
接下来，我们就一起来看一看常见的限流算法，主要包括四种：基于计数的限流算法、基于滑动窗口的限流算法、漏桶算法、令牌桶算法。其实大部分都比较好理解，后面会着重讲解令牌桶算法及其代码实现。

基于计数的限流器

一说到限流，以限制请求次数为例来考虑，我想你一定有一个非常直观的想法，就是直接用计数器来维护一段时间内服务被请求的数量。

比如希望限制系统每分钟处理的请求数量，在 100 次，那我们只需要在内存中维护一个计数器，每次来一个请求，就对这个计数器 +1，另外每过一分钟，也会定时把内存中的计数

器清零。



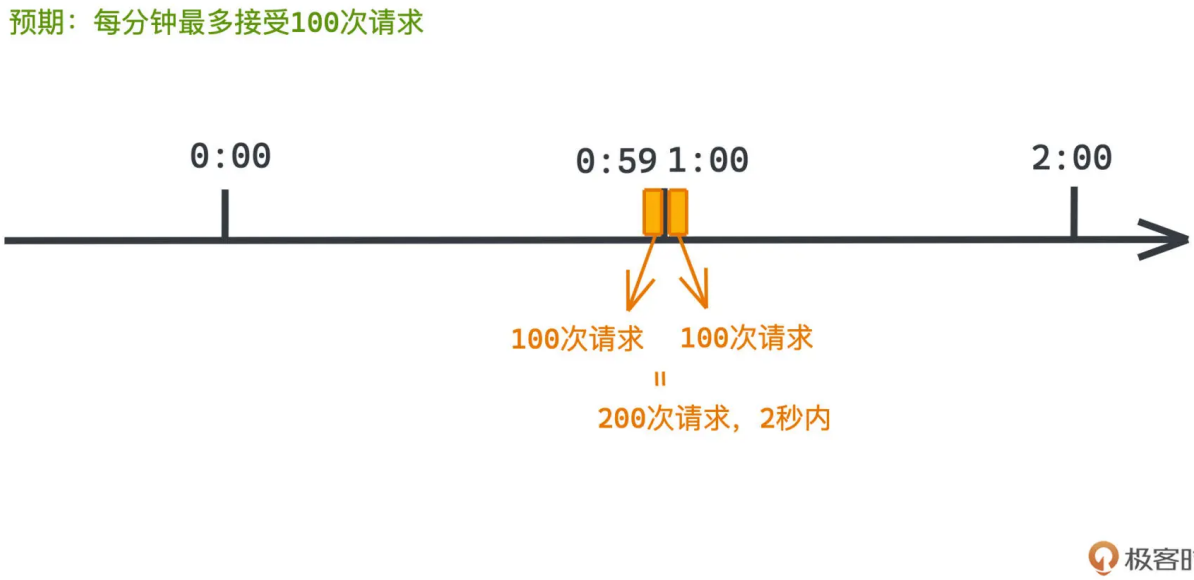
极客时间

当请求来的时候，如果发现计数器的值已经超过 100 次，就会直接拒绝这次请求。显然，这样我们就可以严格地控制每分钟内的请求都不会超过 100 次了。

当然这样只能做到单机的限流，如果希望对某个集群限流，我们自然需要引入一个外部的存储。Redis 就很适合这个场景，每次修改访问计数器的时候，就去修改 Redis，这样我们就可以做到基于计数的分布式限流了。

这个方法，当然是既简单又直观，但有一个很大的问题：我们没法细粒度地控制流量在定时区间内的平滑性，流量依旧可以出现尖刺。比如同样是 1s 内请求 100 次，在这 1 秒的前 10ms 就请求 100 次，和这 1 秒内均匀请求 100 次，两者带来的服务器的瞬时压力是截然不同的。而 100 次每分钟的约束，在流量不均匀分布的时候也很容易遭到破坏。

我们把这个问题称为临界问题，看一个典型的例子：



在每分钟 100 次请求的限流器中，如果在 59 秒和 1 分整这两个时刻内，用户快速请求了 200 次请求，那么在这 2 秒内，用户其实就进行了 200 次请求，而我们建设这个限流器的预期是每分钟最多接受 100 次请求，这显然和我们的预期有着巨大的鸿沟。

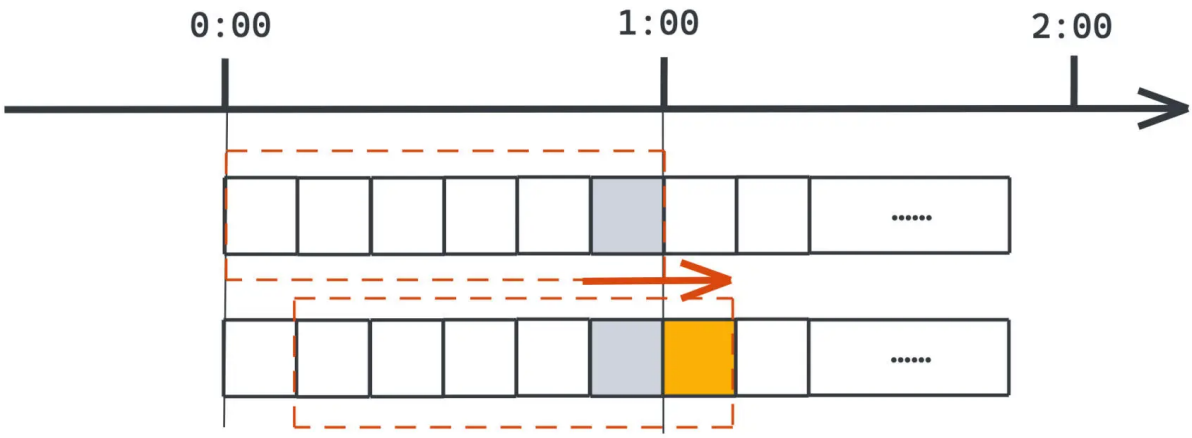
怎么办呢？我们分析一下问题，本质其实就在于这样固定周期清空计数器的方式精度太低，不能区分 1 分钟内均匀请求 100 次，以及只在 1 分钟内很小的一个时间窗口里请求 100 次，这两种情况。

那如何提高精度呢？

基于滑动窗口的限流

TCP 中的滑动窗口思想就可以被我们借鉴了。

在刚才的例子里，既然，以分钟为单位直接统计访问次数粒度太粗，我们可以把统计计数器的区间变小一些，比如以 10 秒为一个区间，每个区间独立统计落在区间内的请求数量。



以10s为一个独立区间

极客时间

现在如何去限制一分钟整体的请求数量呢？

很简单，**我们遍历过去一分钟内每个独立区间，也就是每 10 秒内的计数器的计数总和，这个总和当然就是过去一分钟内全部的请求数量了。当然每次经过 10s，我们也自然需要把整个计数区间往右边移动一格。**在图中的体现就是，当时间经过 1:00 的时候，整个限流器的计数器范围就去掉了最左边 0:00-0:10 的区间，增加了橙色的格子，也就是 1:00-1:10 的区间，这样的过程也就是我们通常所说的“滑动窗口”了。

所以，刚才简单计数的限流算法，它不能正确拒绝的 00:59 和 01:00 连续两秒内请求 200 次的情况，在现在的滑动窗口下显然就会被拒绝了。

在这个思路下，想要进一步提高被控制流量的平滑性，就需要不断增加窗口的精度，也就是缩小每个区间的大小。

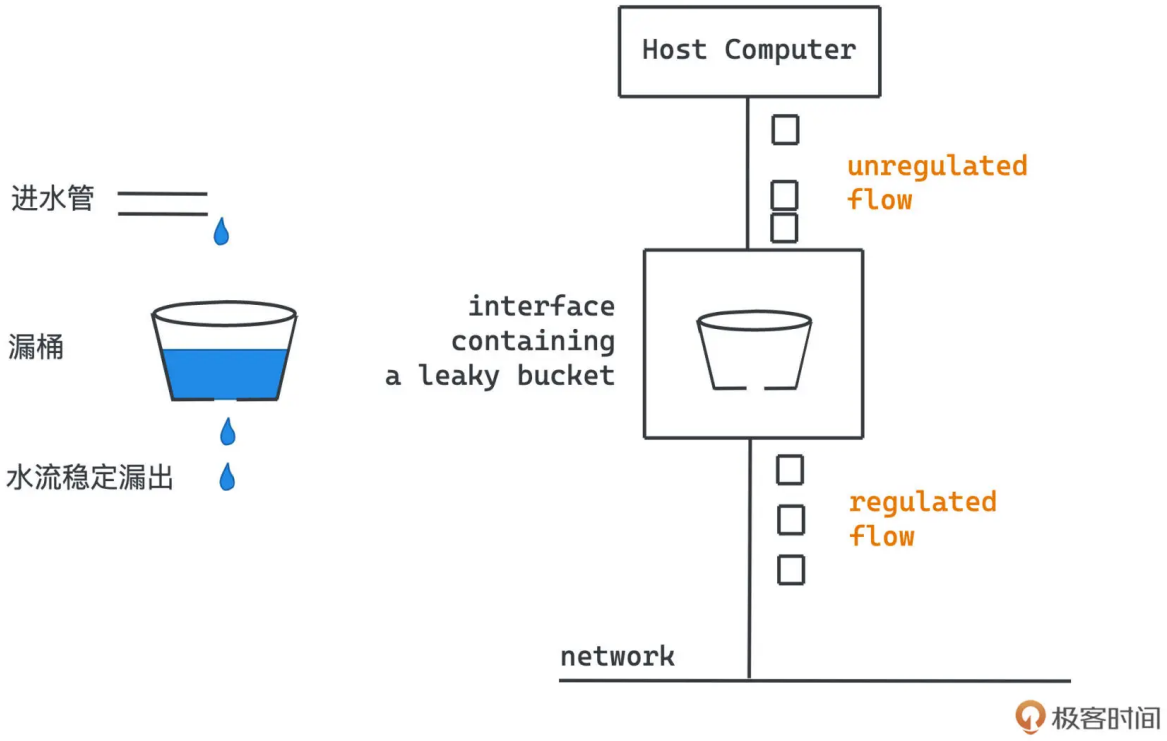
但这样也会带来更多的内存开销，那有没有什么更好的方式可以帮助我们获得理论上更平滑的流量控制能力呢？

漏桶算法

漏桶算法就是这样一种非常平滑的流控方式，它可以严格控制系统处理请求的频率。

漏桶也就是 leaky bucket，名字非常直观，本质就是用计算机模拟一个有漏洞的水桶，只要漏桶中有水存在，水桶的漏洞处就会稳定的有水流漏出。

我们的流量，比如说请求，就像是一个往水桶里不断加入水的水管，水管的流量自然是可以有波动的，可以时快时慢，当漏桶已经被装满的时候，拒绝请求就可以了，就像让水不溢出一样。看这张示意图：



只要我们能让漏洞漏水的消费速度可控、稳定，整个系统只处理漏桶漏出的稳定流量，自然就可以达到限流的效果。

具体如何实现呢？在看到上面对漏桶性质的描述之后，不知道你有没有想到消息队列削峰填谷的特性，这两者的本质其实一样的。所以，漏桶最简单的一种实现正是基于队列。

我们用一个 FIFO 队列模拟水桶，队列的容量就是水桶的最大大小。每来一个请求，就存储到队列中，如果队列满了，就直接拒绝。而另一侧，我们会有一个线程稳定消费队列中的数据，这样，整个系统不管在请求流量多么不稳定的情况下，都可以维持一个非常稳定的流量处理速度。思路非常清晰也很简单，你可以试着实现一下具体的逻辑。

那漏桶算法有没有什么问题呢？

其实从流量平滑的角度来看，已经没什么可挑剔的了，**但计算机的 trade off 无处不在，漏桶这样严格完美的流量限制策略，也使得它完全放弃了应对突发流量的能力**，因为在遇到突发流量时，漏桶的处理和平时并没有区别。

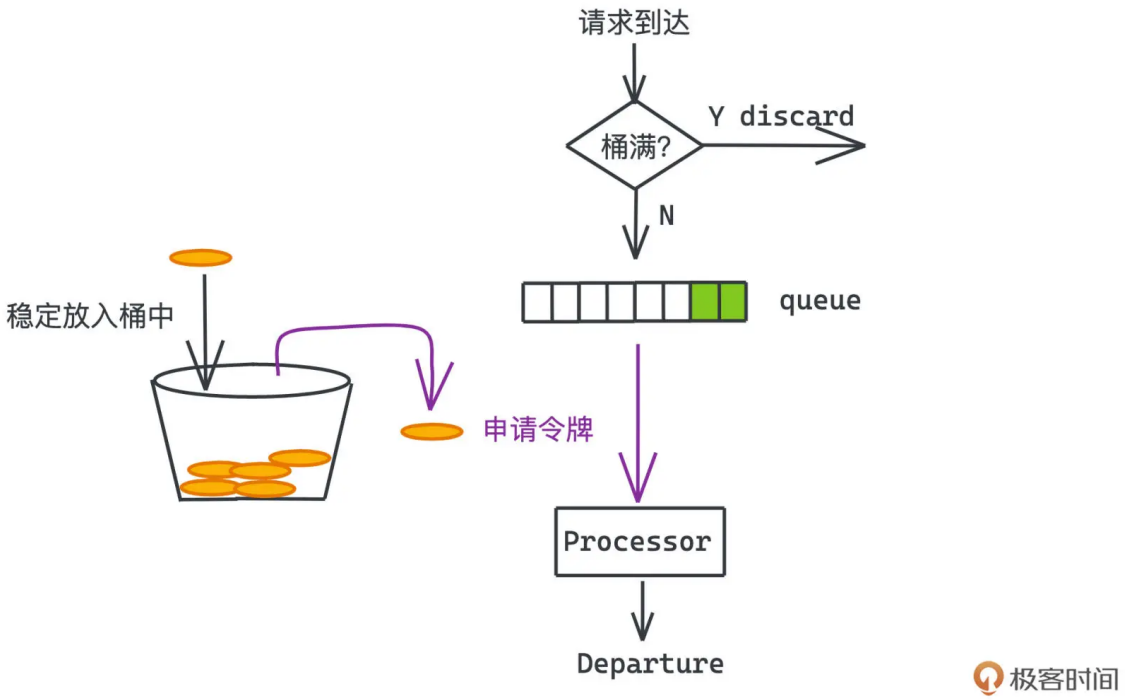
但是大部分服务，短时的处理压力增大，并不会导致整个系统崩溃，很多时候我们也希望在面对突发流量的时候，系统可以稍微提高一下自己的处理速度，以获得更好的用户体验。有没有什么办法呢？

令牌桶算法

这就要提到我们今天最后一个要学习的算法——令牌桶，它也是各个系统中最常见的一种限流策略。

和漏桶一样，令牌桶也采用了桶的模型，不过不同的是，这次我们不再是让系统处理定速从漏桶中流出的流量，而是改成把令牌以稳定的速度放入桶中。桶中令牌数会有一个上限，所以如果令牌桶已经被放满，多余的令牌不会被继续放入了。

每来一个请求，必须先去令牌桶里申请令牌，申请到令牌的请求才能被服务器处理，否则，我们也会拒绝对应的请求。



这个方案流量限制的核心就在于：令牌桶中令牌的数量是有限的。**如果在一段时间内，请求的速度都是高于令牌放入的速度，令牌桶中很快就会没有令牌可用了，服务就会拒绝一部分请求，并保证系统处理的流量在我们的控制范围内。**

好，了解了设计思路，现在我们来看刚才说的面对突发流量，想提高系统短时处理速度的问题。

对比漏桶，令牌桶，在请求流量低的时候，令牌数会慢慢增加直到放满，那么在遇到突发流量时，通常令牌桶内会有一定的令牌数，在这个限度内的请求，即使短时请求速度很快，我们也不会拒绝对应的请求，这就保证了我们在控制流量的同时，也有了一定的突发流量的应对能力。

基于同样的道理，令牌桶自然也可以基于队列实现。用队列存储令牌，每来一个请求就从队列中取出一个令牌，队列为空的时候则拒绝请求，另一边用一个线程稳定的向队列里放入令牌，在队列满的时候停止放入即可。

不过事实上，我们 不需要真的在内存里维护这样占据内存空间的队列，可以采用计算时间的方式来模拟这个过程，核心思路就是：既然我们知道令牌放入的速度，那完全可以通过上一次请求到达的时间和这次请求到达的时间差，判断请求来之前我们又多出来的可用的令牌数量。

用代码实现并不难，大约 50 行左右。你可以参考我写的详细的注释，相关代码在GitHub上也能找到：

复制代码

```
package ratelimit

import (
  "fmt"
  "sync"
  "time"
)

type RateLimiter struct {
  rate   int64 // 令牌放入速度
  max    int64 // 令牌最大数量
  last   int64 // 上一次请求发生时间
  amount int64 // 令牌数量
  lock   sync.Mutex // 由于读写冲突，需要加锁
```

```
}

  // 获得当前时间
func cur() int64 {
  return time.Now().Unix()
}

func New(rate int64, max int64) *RateLimiter {
  // TODO: 检查一下rate和max是否合法
  return &RateLimiter{
    rate:   rate,
    max:    max,
    last:   cur(),
    amount: max,
  }
}

func (rl *RateLimiter) Pass() bool {
  rl.lock.Lock()
  defer rl.lock.Unlock()

      // 距离上一次请求过去的时间
  passed := cur() - rl.last
  fmt.Println("passed is: ", passed)

      // 计算在这段时间里 令牌数量可以增加多少
  amount := rl.amount + passed*rl.rate

      // 如果令牌数量超过上限；我们就不继续放入那么多令牌了
  if amount > rl.max {
    amount = rl.max
  }

      // 如果令牌数量仍然小于0，则说明请求应该拒绝
  if amount <= 0 {
    return false
  }

      // 请求被放行则令牌数-1
  amount--
  rl.amount = amount
      // 更新上次请求时间
  rl.last = cur()

  return true
}
```

总结

我们学习了四种常见的限流算法：基于计数的限流算法、基于滑动窗口的限流算法、漏桶算法、令牌桶算法。

从递进的顺序来看，你很可能会觉得令牌桶算法要比前面的算法都好，而时间窗口是一种不够优秀的算法。但事实上，这几种算法其实都有各自的长处。**计算机的世界里到处都是 trade off，魔法并不存在，要始终铭记在权衡方案时，如果我们在某些方面获得了一些好处，那就一定要警觉在另一些方面是否付出了一些代价，以及代价是否可承受。**

比如，滑动窗口，虽然会产生一定的尖峰，且需要比较大的内存开销，但是一旦请求来了，要么会被立刻拒绝，要么会被立刻响应，不会有太大的延时。

而漏桶会维护一个队列，导致没有被拒绝的请求，真正被执行的时间可能会比较靠后，这就可能产生较大的时延，在时间敏感的场景下，漏桶就不太合适。在一些令牌桶的实现中，也会有一个队列缓冲部分没有令牌的请求，这些请求的处理也同样会产生比较大的时延。所以，令牌桶和漏桶其实更适合后台任务这样可以接受一定时延的场景。

在不同的场景下，我们可以选择不同的限流实现，当然在生产环境中相比于自己动手实现，采用成熟的中间件或者类库，当然是更稳妥的选择。Google 的 Guava 库就提供了基于令牌桶的实现，Nginx 和 Resty 这样的网关代理组件也都有相关的实现，可以择优选用。

课后习题

请你实现一下漏桶算法，并思考除了用 FIFO 队列的方式，还有没有什么其他内存使用更少的实现方式呢？

欢迎你在评论区留下你的代码和思考，一起参与讨论，如果觉得这篇文章对你有帮助的话，也欢迎转发给你的朋友一起学习。我们下节课见～

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 32 | 时间轮：Kafka是如何实现定时任务的？

下一篇 34 | 前缀树：Web框架中如何实现路由匹配？

更多学习推荐

精选留言 (3)

写留言

那时刻
2022-03-10

我想漏桶算法，可以通过一个计数器来计算目前in flight的数据包数量，如果来了数据包则加一，数据包处理完减一，如果计数器超过limit，则限流返回。

peter
2022-03-10

请教老师两个问题： Q1：云产品例子中： A 是哪个公司的产品？阿里？或者仅仅是个示例？ B 该产品是个消息队列吗？ C 5000连接 / 最高 10,000TPS，连接数和TPS有数量关系吗？ 此处好像是2倍关系。另外，连接数是指同时连接数吧。 Q2：漏桶消费线程怎么实

现“稳定消费”？定时消费吗？ 令牌桶的放令牌线程，也是“稳定放入令牌”，也是定时放入令牌吗？

展开

Paul Shan

2022-03-10

漏桶算法也可以用一个计数器来维护当前的请求数目，新来一个请求，如果计数器到达上限就拒绝请求，如果没有到达上限，计数器自增后把请求交给具体的处理器，请求被处理完毕的时候计数器自减。